2007

Optimization for software release and crash

Tanvir Khan
Louisiana State University and Agricultural and Mechanical College, tkhan1@lsu.edu

OPTIMIZATION FOR SOFTWARE RELEASE AND CRASH

A Thesis

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Master of Science in Engineering Science

in

The Interdepartmental Program in Engineering Science

by
Tanvir Khan
B.S., Louisiana State University – Baton Rouge, 2004
May, 2007

ACKNOWLEDGEMENTS

I would like to thank my dad Abu Haniff Khan, mom Nazmun A. Khan, sister Sharmin Khan, and fiancé Zariat Afrin, for their help, guidance, and support throughout my educational career. It would not have been possible for me to achieve this academic accomplishment without them.

I would also like to thank Dr. David Constant, Vicki Hannan, and other staffs of the College of Engineering Dean's Office and the Graduate School for their exceptional administrative assistance and guidance.

Thanks to Dr. Donald L. Crumbley for helping me gain valuable experience in the field of forensic and investigative accounting, Dr. Gerald M. Knapp for his teaching of the most recent computer languages and advanced technologies, and Dr. Xiaoyue Jiang for introducing me to the area of reliability engineering, and giving me the opportunity to conduct research in that field. Their valuable teachings, advices, and cooperation have truly made my graduate school experience skillful, and memorable.

Lastly, and of course most importantly, I would like to specially thank Dr. Junfang Yu, and Dr. Dan B. Rinks, for their time, patience, and extraordinary support as committee members.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Software testing is a process to detect faults in the completeness and quality of developed computer software. Testing is a key process in assuring quality by identifying defects in software, and possibly fixing them, before it is delivered to end-users. A major decision to make during this software testing is, to determine whether to continue testing and eventually releasing the software, or when to stop the test and ‘crash’ it. Such a decision needs to be made to optimally balance the tradeoff between the cost of development and the reliability of the software. In this paper, a new optimal strategy is developed based on a conditional non-homogeneous Poisson process (Conditional-NHPP) on a continuous time horizon to determine when the optimal time is to release or crash the software.

1. INTRODUCTION

With the dawn of a new era, the computer age, computers and the software running on them are playing an essential role in our daily lives. Almost all analog, mechanical appliances that we used to operate have been replaced by digital equipments, run by CPUs and software. From cars to critical defense equipments, virtually everything uses sophisticated electrical systems and smart chips today. Unlike in the past when mechanical components were primarily used, electrical systems now use embedded software. With the advancement in web technology, many hardware based systems are substituted by software applications. The growth of software development and web technology has been enormous since the Internet revolution. As Microsoft CEO Steve Ballmer (2006) puts it, we have now entered “a new software era”.

Software applications are used in many critical devices – airplanes, heart pace-makers, radiation therapy machines, etc. A software error in such machines can claim people’s lives. With processors and software saturating the safety critical embedded world, the reliability of software is simply a matter of life and death (Pan (1999)).

1.1 Software Reliability

Software reliability is the likelihood of successful operation of software for a predetermined period of time in a specified environment. Software is considered to have performed a successful operation, when it functions completely as expected, without any failure. Software that fails less often is considered to have higher quality than software that fails frequently.

Software unreliability is a consequence of unexpected results of software operations. Even comparatively small software programs can have a large combination of inputs and states that are impracticable to test thoroughly. Software reliability engineering must take into account that restoring software to its original state only works until the same combination of inputs and

states results in the same inadvertent result (Wikipedia (2007)). For two identical copies of the same software, the reliability may be different if they are used under different operational conditions (Musa and Okumoto (1984)). These factors essentially exhibit the characteristic of randomness in software reliability engineering.

1.2 Software Reliability Growth Models

The objective of software reliability testing is to determine probable problems with the software design and implementation as early as possible to assure that the system meets its reliability requirements. Several hundreds of statistical models used in software reliability testing have been developed over the years. Among the models, software reliability growth models, also knows as SRGMs (Lyu (1996), Xie (1991), Tohma et. al (1989), Musa et. al. (1987), Ohba (1984), Yamada et. al. (1984), and Goel and Okumoto (1979)), are most widely used.

A SRGM is a useful mathematical tool to describe the failure-occurrence or fault-detection phenomena in the software testing phase, and to assess software reliability quantitatively (Inoue and Yamada (2007)). SRGMs can be used to characterize the dynamics of the testing process (e.g., number of initial faults, the software reliability, etc.), and to predict possible failure pattern (e.g., failure intensity, meantime-interval between failures, etc.) (Huang et. al. (2003)).

De-eutrophication Model

The de-eutrophication model developed by Jelinski and Moranda in 1972 is one of the first SRGMs for assessing software reliability, that has become the basis for much of the research thereafter. The model is based on a hypothesis that software contains a finite number of bugs, and the inter-failure time is exponential with intensity proportional to the number of remaining bugs. Each bug in the software causing a failure is instantly removed, and therefore the number of bugs remaining is reduced by one. Schick and Wolverton in 1978 examined its applicability to

the error-finding process with actual data.

Several methods that determine the parameters of the de-eutrophication model have demonstrated that the maximum likelihood count of the defects in the software is a favorable approach for software testing (Forman and Singpurwalla (1977)).

The de-eutrophication model has been used in decision making. In (Shanthikumar and Tufekci (1983)), termination of software testing occurs when the detected number of defects exceeds a limit, which is based on the rationale that the detection of a large number of defects reduces the number of remaining defects in the software.

Further modifications of the de-eutrophication model include a randomized initial number of defects mapped to a Poisson distribution with another random parameter λ, characterized by a gamma distribution (Dalal and Mallows (1988)). Termination is triggered when the number of defects falls considerably below the initially estimated number (Zheng (2002)).

A software reliability model has been studied by (Ross (1985)), where a fixed number of bugs (or failure) is considered, and that each of these bugs independently causes failure of software following an exponential distribution. The number of bugs in software and the rate of failure occurrence are both primarily unknown. The optimal stopping time is obtained when the failure rate is less than an acceptable predetermined failure rate. The model has further been modified (Yamada and Osaki (1985)) by evaluating the total average cost and software reliability simultaneously. The total cost in this model was minimized within the constraint of software reliability.

Models assessing the effects of reliability and cumulative costs of software testing have demonstrated that expected average costs of software testing can be reduced by restricting the scope of testing procedures (Thayer et. al. (1976)).

In this research, a generalization of the de-eutrophication model, *double* Bayesian method, will be used, where the number of bugs is a Poisson random variable with a random parameter. The resulting model is known as a conditional non-homogeneous Poisson process (Conditional-NHPP).

1.3 Optimal Stopping Problem

The use of SRGMs to depict software reliability provides a statistical foundation to establish optimal stopping time for software testing, which is a key decision problem in software engineering. One of the major issues leading to software maintenance cost overruns and customer rejection is due to insufficient testing time (Humphrey (1989)). The software testing process is both time-consuming and costly. However, much more time and cost could be spent in maintenance of software later due to fixes of bugs not discovered during the testing phase.

The optimal stopping problem includes three possible actions that can be chosen at any time: continue testing, release or crash the software. Releasing the software means that the software being tested is reliable enough for the testing procedure to be stopped, and that it can be made available to users. Crashing the software means simply to stop the testing process and abandon the software if it is found to be too unreliable to be released. The optimal stopping problem can consequently be used to determine the most favorable time to end testing, and crash or release the software. In addition to the optimal releasing problem (also refer to (Forman and Singpurwalla (1979), Okumoto and Goel (1980), Koch and Kubat (1983), Yamada et. al. (1984), Kapur and Garg (1989), and Dalal and Mallows (1990))), whether software should be released at all can be incorporated with the decision process.

The terminology of “crashing” in different contexts may possess different meanings. For example, in project management, crashing refers to a strategy of accelerating the process by adding new resources to the development (Biafore (2006)). In another context, software crash

refers to an unexpected or a sudden failure of a software program or operating system. The term "crash" used in this research signifies abandonment, i.e., during software testing, crashing software means terminating the test and abandoning the software.

Release policies can be classified into two categories: static or dynamic. In case of static release policies, the release time of software is determined before testing has begun, and is kept unchanged throughout the testing phase. The release time is independent of any discrepancy due to data collected during the testing phase (Jiang et. al (2005)). Under dynamic release policies, there is no preset release time. The release time is dynamically determined from failure statistics obtained during testing.

Crash policy can be stated as such that if the software during testing is found to be exceedingly unreliable to release, then abandoning it instead of continuing to test is a cost-effective option. Since time is a critical factor in today's competitive software industry, in many cases, the crashing option over prolonged testing is a more economical and realistic preference.

In this research, integrated optimal release/crash policies for software testing will be developed, based on a conditional non-homogeneous Poisson process on a continuous time horizon. The optimal policies will include crash and release options based on monotonicity, cost structure and number of bugs detected during testing.

2. PROBLEM FORMULATION

The decision when is the best time to stop software testing can be derived from the optimal crash and release policies. A mathematical model is proposed in this section to determine the optimal crash and release strategies for computer software testing, based on optimal stopping formulation. The model is an enhancement of a previous research (Jiang et. al (2005)), where the release time for software was predetermined, and kept constant throughout the testing phase.

2.1 Notations

The following is a list of notations used in the formulation of the mathematical model:

N	Number of bugs detected.
$N(t)$	Number of bugs detected within $[0, t]$.
X	Random parameter in the intensity function of debugging process.
Ω	The sample space of X.
$g(t)$	Probability density function of debugging time; $G(t) = \int_0^t g(t)dt$, $(X \leq U)$.
$p(x)$	Probability density function of X.
T_N	Deadline time for releasing the software.
T_i	Crashing threshold for the occurrence of the i^{th} bug.
S_i	Time of the detection of the i^{th} bug.
$\mathcal{F}_t$	Completed filtration of debugging process.
$f(t)$	Cost of testing during period $[0, t]$. Assume $f(t)$ is differentiable.
C_p	Penalty cost for not delivering software by the deadline.
C_r	Cost of fixing one bug before release.
C_R	Cost of fixing one bug after release.
$TC(t)$	Total expected cost when software is released at t.
$RC(t)$	Total expected cost of fixing all remaining bugs after released at t.
$EX_n(t)$	$E(X\|N(t) = n)$: Mean of X given the number of bugs by t is n.
$\varphi_n(t)$	$C_R EX_n(t)\bar{G}(t)$: Expected cost after release.
$\gamma_n(t)$	$EXn(t)g(t)$: Rate of bug occurrence.
$\bar{F}_n(t)$	$\exp\{-\int_0^t \gamma_n(s)ds\}$: Survival function of the next failure time.
$\bar{F}_n(t\|s)$	$\bar{F}_n(t)/\bar{F}_n(s)$: Survival function of the residual failure time.
$V_n(t)$	Total cost if the optimal policy is chosen, starting at (n, t).
$W_n(t)$	Difference between the total cost and the expected cost after release, or savings $[V_n(t) - \varphi_n(t)]$.
U	Uniform distribution with parameters $[U_{MIN}, U_{MAX}]$.

2.2 Mathematical Model

The following mathematical model is proposed to establish the optimal release and crash policies, based on optimal stopping formulation:

1. Time horizon:

   The system runs on a continuous time horizon $[0, \infty]$.

2. Dynamics:

   The dynamics of the testing process is a conditional non-homogeneous Poisson process (NHPP) such that,

   i. the detection times of individual bugs are randomly distributed with probability density function $g(t)$, which is assumed to decrease over time,

   ii. X is an unobservable random variable with distribution $p(x)$,

   iii. conditioning on $X = x$, the number of bugs detected in $[0, t]$, $N(t)$, follows a NHPP with intensity $\gamma(t) = xg(t)$.

3. Action set:

   There are three possible actions available to choose from at any time: continue testing, release or crash the software.

4. Cost structure:

   C_p, C_r, C_R, and $f(t)$ represent the penalty cost for crashing the software, cost of fixing one bug during testing, cost of fixing one bug after releasing, and the cost of testing during the period $[0, t]$, respectively. Assume, for the optimal policy,

   $$f(t) = h(G(t)), \tag{1}$$

   where h is convex.

5. Objective criterion:

   The objective is to minimize the total cost of software associated with testing, debugging

and/or crashing.

The rationale behind the assumptions in the formulation of the mathematical model is that, the number of bugs detected within $[0, t]$, $N(t)$, forms a conditional NHPP with intensity function $xg(t)$ when $X = x$. This is the case when the number of initial bugs follows a Poisson distribution with parameter X. Each failure is followed by one bug, and the times at which individual bugs are detected are independent, and identically distributed with the probability density function $g(t)$. The detected bug is corrected or removed, no new bugs are introduced, and the correction of a bug takes an insignificant amount of time (Kuo and Yang (1996)).

In (Zheng (2002)), the detection time follows an exponential distribution with parameter μ, where $g(t) = \mu e^{-\mu t}$. Another variation of the model has been investigated (Dalal and Mallows (1988)), where X is gamma distributed with parameter (α, β).

2.3 The Cost Process

The software testing process can be lengthy and expensive. It can involve additional testing time, leading to increased cost in software maintenance, if a large number of bugs are not discovered during testing to ensure the reliability of the software. It is thus imperative that optimal policies be applied to attain the best possible time to cost effectively crash or release the software. The following cost process model characterizes the total cost of the testing process.

In this model, a system being considered is tested for a duration *t* days. Suppose, the total number of bugs detected within $[0, t]$ is $N(t)$, and $S_1, \ldots, S_n, \ldots$ are the arrival times of the detected bugs. The bug detection process $\{S_1, \ldots, S_{N(t)}\}$ generates a filtration, denoted by $\mathcal{F}_t$. The total cost, $TC(t)$, for the software system being tested for the duration *t* and to be released at t, has the following expression:

$$TC(t) = f(t) + C_r N(t) + C_R E X_n(t) \bar{G}(t) \tag{2}$$

In (2), $f(t)$ is the cost of testing related to the duration, $C_r N(t)$ is the cost of testing

(product of the cost of fixing one bug before release and the total number of bugs found during testing) related with repairing of the bugs, and $C_R EX_n(t)\bar{G}(t)$ is the expected cost of fixing the remaining bugs after release.

If the testing is stopped at time t, the expected cost due to the unknown number of bugs remaining in the software needs to be added. Let $RC(t)$ denote the probable cost of fixing all residual bugs after release. Then,

$$RC(t) = C_R EX_n(t)\bar{G}(t). \tag{3}$$

From here on, $E[X|N(t) = n]$ will be denoted as $EX_n(t)$, $RC(t)$ as $\varphi_n(t)$, and the failure intensity $EX_n(t)g(t)$ as $\gamma_n(t)$.

The minimization problem as an optimal stopping problem is to find $\{\mathcal{F}_t\}$ – stopping time τ^*, if exists, such that

$$E\big(TC(\tau^*)\big) = inf_\tau(E\big(TC(\tau)\big)). \tag{4}$$

In order to portray the total cost function, $TC(t)$ explicitly, the stopping time τ needs to be characterized first. The stopping time of jumping process τ (Jiang and Makis (2003)) can be denoted as

$$\tau = \sum_{i=0}^{\sigma-1}(S_{i+1} \wedge T_i - S_i)\prod_{j=0}^{i-1} I_{\{S_{j+1}<T_j\}} \tag{5}$$

$$\equiv \tau_\sigma \wedge \tau_T.$$

In (5), σ is a stopping time with respect to the discrete filtration

$$\mathcal{H}_n \equiv \mathcal{F}_{S_n},$$

$$\tau_\sigma = S_\sigma \tag{6}$$

is the $\{\mathcal{F}_t\}$ – stopping time at jump points, and

$$\tau_T = \sum_{i=0}^{\infty}(S_{i+1} \wedge T_i - S_i)\prod_{j=0}^{i-1} I_{\{S_{j+1}<T_j\}} \tag{7}$$

is the $\{\mathcal{F}_t\}$ – stopping time between jump points with probability one. Here, $\{T_i \geq S_i\}$ is adapted to $\mathcal{F}_{s_i}$.

From the monotonicity of the releasing cost $\varphi_n(t)$, notice that it is unreasonable to crash between jumps or to release at jumps. From this observation, the following restriction can be applied.

Restriction A

The crash option is associated with τ_σ, and the release option with τ_T, for any stopping time τ. In other words, release is not considered at jumps, and crash is not considered between jumps.

Using the restriction above, the total cost for the system stopped at τ, $TC(\tau)$ can be represented by the following expression.

Lemma 2.1

$$TC(\tau) = \varphi_0(0) + \sum_{i=0}^{\sigma} A_i \prod_{j=0}^{i-1} I_{\{S_{j+1}<T_j\}} \tag{8}$$

where for $i < \sigma$,

$$\begin{aligned} A_i = \; & -\varphi_i(S_i) + \varphi_i(S_{i+1} \wedge T_i) + I_{\{S_{i+1}<T_i\}}[C_r + \varphi_{i+1}(S_{i+1}) - \varphi_i(S_{i+1})] \\ & + f(S_{i+1} \wedge T_i) - f(S_i). \end{aligned} \tag{9}$$

For $i = \sigma$,

$$A_\sigma = \left(C_p - C_r I_{\{\sigma>0\}}\right) - \varphi_\sigma(S_\sigma). \tag{10}$$

Proof.

Using Restriction A, for all possible sample paths, when $\sigma = 0$,

$$\begin{aligned} TC(\tau) &= \varphi_0(0) + A_0 \\ &= \varphi_0(0) + \left(C_p - C_r I_{0>0\}}\right) - \varphi_0(S_0) \\ &= C_p \; . \end{aligned} \tag{11}$$

$TC(\tau)$ at $\sigma = 0$ coincides with the cost of crashing the system at $S_0 = 0$.

When system testing stops at $S_n < t < S_{n+1}$ for releasing, the total cost is

$$TC(t) = f(t) + C_r N(t) + \varphi_n(t). \tag{12}$$

This situation relates to the case where $\sigma > n$, $T_n = t$, and $S_{i+1} > T_i$.

Recall, from (17), the total cost for the system stopped at τ, $TC(\tau)$,

$$TC(\tau) = \varphi_0(0) + \sum_{i=0}^{\sigma} A_i \prod_{j=0}^{i-1} I_{\{S_{j+1} < T_j\}}$$

Then,

$$\begin{aligned} TC(\tau) &= \varphi_0(0) + \sum_{i=0}^{n-1} A_i \\ &= \varphi_0(0) + \sum_{i=0}^{n-1} [-\varphi_i(S_i) + \varphi_{i+1}(S_{i+1}) + C_r + f(S_{i+1}) - f(S_i)] \\ &\quad + [-\varphi_n(S_n) + \varphi_n(T_n) + f(T_n) - f(S_n)] \\ &= nC_r + \varphi_n(T_n) + f(T_n)\cdot \end{aligned} \tag{13}$$

When the system testing stops at $t = S_n\,,\ n > 0$ for crashing,

$$\begin{aligned} TC(\tau) &= \varphi_0(0) + \sum_{i=0}^{n-1} [-\varphi_i\,(S_i) + \varphi_{i+1}(S_{i+1}) \\ &\quad + C_r + f(S_{i+1}) - f(S_i)] + [\left(C_p - C_r\right) - \varphi_n(S_n)] \\ &= (n-1)C_r + C_p + f(S_n). \end{aligned} \tag{14}$$

The debugging cost is $(n-1)C_r$ instead of nC_r as the last detected bug, which need not be if the software is crashed immediately.

2.4 Problem Reduction

The original optimal stopping problem can be further simplified from the above representation of *TC*(τ) using semi-martingale decomposition method.

$$TC(\tau) = \varphi_0(0) + \sum_{i=0}^{\sigma} A_i \prod_{j=0}^{i-1} I_{\{S_{j+1} < T_j\}}$$

$$= \varphi_0(0) + \sum_{i=0}^{\sigma} E\left(A_i \middle| \mathcal{F}_{S_i}\right) \prod_{j=0}^{i-1} I_{\{S_{j+1} < T_j\}} + M_\tau \,, \tag{15}$$

where M_τ is the martingale part with respect to stopping time τ with $M_0 = 0$.

The Optional Stopping Theorem (Elliot (1982)) can be applied, due to the boundedness of τ:

$$E(TC(\tau)) = \varphi_0(0) + \sum_{i=0}^{\sigma} E\left(A_i \middle| \mathcal{F}_{S_i}\right) \prod_{j=0}^{i-1} I_{\{S_{j+1} < T_j\}} + EM_\tau$$

where, $EM_\tau = 0$. This implies the optimal stopping time for the progressive part without the loss of optimality.

Therefore, the cost process

$$E\left(TC(\tau)\right) = \varphi_0(0) + \sum_{i=0}^{\sigma} E\left(A_i | \mathcal{F}_{S_i}\right) \prod_{j=0}^{i-1} I_{\{S_{j+1} < T_j\}} \tag{16}$$

defines a dynamic system (Jiang and Makis (2003)) with initial state $(n, t) = (0,0)$:

$$E\left(TC(\tau)\right) = \varphi_n(t) + \sum_{i=0}^{\sigma-1} \int_{S_i}^{T_i} \bar{F}_i(s|S_i)[\gamma_{n+i}(s)(C_r - C_R) + f'(t)]ds$$

$$\prod_{j=0}^{i-1} I_{\{S_{j+1} < T_j\}} - (\varphi_{n+\sigma}(S_\sigma) - \left(C_p - C_r I_{\{\sigma>0\}}\right)) \prod_{j=0}^{\sigma-1} I_{\{S_{j+1} < T_j\}} \tag{17}$$

Then, the optimal expected total cost, $V_n(t)$ for the system with initial state (*n, t*) becomes

$$V_n(t) = inf_{\{\sigma | T_{n+i}\}} \{\phi_n(\mathrm{t}) - \sum_{i=0}^{\sigma-1} \int_{S_i \wedge \mathrm{T_{n+i}}}^{\mathrm{T_{n+i}}} \bar{\mathrm{F}}_{n+i}\,(s|S_i)\,[\gamma_{n+i}\,(s)\,(C_R - C_r) - f'(t)]\; ds$$

$$- [\phi_{n+\sigma}\,(S_\sigma) - \left(C_p - C_r I_{\{\sigma>0\}}\right)]\; I_{S_\sigma < T_n}\} \,. \tag{18}$$

This system holds the Markov property with respect to its initial state (n, t), and hence dynamic programming approach can be applied to solve the optimization problem. The solution to the dynamic programming problem consequently becomes the optimal policy among the whole $\mathcal{F}_t$- stopping time class.

3. STRUCTURAL PROPERTY OF THE OPTIMAL POLICY

During software testing, any of the three decisions need to be taken: to terminate testing and to crash the software, to continue testing for a longer period, or to release the software after sufficient testing. Since cost and time are critical in today's software business, it is important to optimally make a decision whether to crash or release the software once the most favorable time is reached. The optimal time is determined by failure statistics such that, if an increasing number of bugs are detected within a specified testing period, the crash option is considered optimal. Conversely, if fewer bugs are detected over the continuous time horizon, the software is considered reliable enough to be released.

The time that defines when it is most advantageous to crash or release the software is based on optimal policies. In this section, dynamic programming method will be used to derive such policies.

3.1 Dynamic Equation

For a dynamic system that allows both crash and release options between jumps, we can denote the cost function $W_n(t)$ with initial state (n, t). Recall $V_n(t)$ (18), the cost if the optimal policy is chosen starting at (n, t). Then, the savings (or gain) can be represented as

$$W_n(t) = V_n(t) - \varphi_n(t). \tag{19}$$

For $V_n(t) \leq C_p$,

$$V_n(t) - \varphi_n(t) = Min_T\{-\frac{1}{\bar{F}_n(t)}\int_t^T \bar{F}_n(s)\big\{[\gamma_n(s)(C_R - C_r) - f'(s)] - \gamma_n(s)\big[V_{n+1}(s) \wedge (C_p - C_r) - \varphi_{n+1}(s)\big]\big\}ds\}. \tag{20}$$

$$W_n(t) = Min_T\{-\int_t^T \bar{F}_n(s|t)\{[\gamma_n(s)(C_R - C_r) - f'(s)] - \gamma_n(s)[W_{n+1}(s) \wedge (C_p - C_r - \varphi_{n+1}(s))]\}ds\}. \tag{21}$$

Since $W_n \leq 0$ always, we consider

$$|W_n(t)| = \int_t^{T_n^*} \bar{F}_n(s|t)\{[\gamma_n(s)(C_R - C_r + |W_{n+1}(s)| \vee (\varphi_{n+1}(s) - (C_p - C_r))) - f'(s)] \quad (22)$$

The monotonicity of $W_n(t)$ and $|W_n(t)|$ determines the forms of optimal crash and release policies.

3.2 Optimal Release Policy

Intuitively, the optimal release policy represents the most favorable time to release the software. The policy is simply to release the software at the optimal release time, denoted by T_n^*.

By assumption from (1), the cost structure $f(t) = h(G(t))$, where h is convex.

Therefore,

$$f'(t) = h'\big(G(t)\big)g(t)\ , \quad (23)$$

and

$$\frac{f'(t)}{\gamma_n(t)} = \frac{h'\big(G(t)\big)g(t)}{EX_n(t)g(t)} = \frac{h'\big(G(t)\big)}{EX_n(t)} \quad (24)$$

is increasing as t.

The optimal release time for the n-th bug T_n^* satisfies

$$\begin{aligned} f'(s) &= \gamma_n(s)(C_R - C_r - [W_{n+1}(s) \wedge (C_p - C_r - \varphi_{n+1}(s))]) \\ &= \gamma_n(s)(C_R - C_r + |W_{n+1}(s)| \vee [(\varphi_{n+1}(s) - (C_p - C_r)]. \end{aligned} \quad (25)$$

Once we show that $|W_{n+1}(t)| \downarrow t$, (25) has a unique solution. Notice that with the uniqueness of T_n^*, we know that the release time for the n-th bug is independent to $t = S_n$, when the n-th bug is detected.

The monotonicity $|W_n(t)| \uparrow n \downarrow t$ guarantees that the optimal releasing policy is a control-limit policy with $T_n \uparrow$ for any n. Hence, the following lemma is established to prove that $|W_n(t)| \uparrow n \downarrow t$.

Lemma 3.1 $|W_n(t)| \uparrow n \downarrow t$.

Proof.

To prove that $|W_n(t)| \uparrow n \downarrow t$, we need the following result from stochastic comparison (Zheng (2002)):

Let $X \geq_{s.t.} Y$, and $h(x)$ decreasing. Then,

$$E[h(X) \leq E[h(Y)]. \tag{26}$$

The above result is applied to prove $|W_n(t)| \uparrow n$, and $|W_n(t)| \downarrow t$. Based on mathematical induction, the following assumptions are made:

$$|W_{n+1}(t)| \uparrow n, \text{ and } |W_{n+1}(t)| \downarrow t.$$

1. Proof of $|W_n(t)| \uparrow n$.

a) Let $h_n(s) = \left(C_R - C_r + |W_{n+1}(S)| \vee \left(\varphi_{n+1}(s) - \left(C_p - C_r\right)\right)\right) - f'(s)/\gamma_n(s)$ if $s \leq T_n^*$, and $h_n = 0$ otherwise. It is decreasing as s and increases as n, given the induction assumption from $(n+1)$.

b) $X_n \geq_{s.t.} X_{n+1}$.

By (a) and (b), together with the stochastic comparison result, we have

$$\begin{aligned} |W_n(t)| &= \int_t^{T_n^*} f_n(s|t) h_n(s) ds \\ &\leq \int_t^{T_n^*} \bar{F}_n(s|t) h_{n+1}(s) ds (h_n(t) \uparrow n) \\ &\leq \int_t^{T_n^*} f_{n+1}(s|t) h_{n+1}(s) ds \qquad (27) \\ &\leq \int_t^{T_{n+1}^*} f_{n+1}(s|t) h_{n+1}(s) ds \\ &= Max_T \{ \int_t^T \bar{F}_{n+1}(s|t) \{ [\gamma_{n+1}(s)(C_R - C_r \end{aligned}$$

$$+|W_{n+2}(s)|\vee(\varphi_{n+2}(s)-(C_p-C_r)))-f'(s)]\}ds\}$$

$$=|W_{n+1}(t)|.$$

Since $|W_n(t)| \le |W_{n+1}(t)|$, $|W_n(t)| \uparrow n$.

2. Proof of $|W_n(t)| \downarrow t$.

For $t_1 < t_2$, clearly, $X_n(t_2) \ge X_n(t_1)$.

$$|W_n(t_2)| = \int_t^{T_n^*} f_n(s|t_2)h_n(s)ds$$

$$= \int_0^{T_n^*-t_2} f_n(s-t_2|t_2)h_n(s+t_2)ds$$

$$\le \int_0^{T_n^*-t_1} f_n(s-t_2|t_2)h_n(s+t_2)ds \qquad (28)$$

(longer integration interval, and positive integrand)

$$\le \int_0^{T_n^*-t_1} f_n(s+t_2|t_2)h_n(s+t_1)ds(h_n(t) \downarrow t)$$

$$\le \int_0^{T_n^*-t_1} f_n(s+t_1|t_1)h_n(s+t_1)ds$$

$$= \int_{t_1}^{T_n^*} f_n(s|t_1)h_n(s)ds(X_n(t_2) \ge X_n(t_1)$$

$$=|W_n(t_1)|.$$

Since $|W_n(t_2)| \le |W_n(t_1)|$, $|W_n(t)| \downarrow t$.

As all the monotonicity properties of $|W_n(t)|$ have been proved, we have control-limit form of the optimal release policy with $\{T_n^*\}$ increases in n.

3.3 Optimal Crash Policy

The optimal crash policy is determined by the dynamic equation (20). In fact, based on the optimal stopping for Markov processes (Chow et. al. (1991)), the optimal crash

decision is made for S_σ, σ being the optimal crashing policy, if and only if the optimal cost function

$$V_\sigma(S_\sigma) - (C_p - C_r I_{\{\sigma>0\}}) \leq 0. \tag{29}$$

Based on (29), the optimal crash policy can then be interpreted as the following:

At the jump point $t = S_n,\ n \geq 1$, the remaining system has a minimal cost $V_n(t)$. When the cost is higher than $(C_p - C_r)$, fixing the current defect with cost C_r, and continuing to test will cost more than simply crashing the software. When $\sigma = 0$, the software should not enter the testing stage if $V_0(t) = C_p$. The optimal crash policy can be illustrated as in Figure 1.

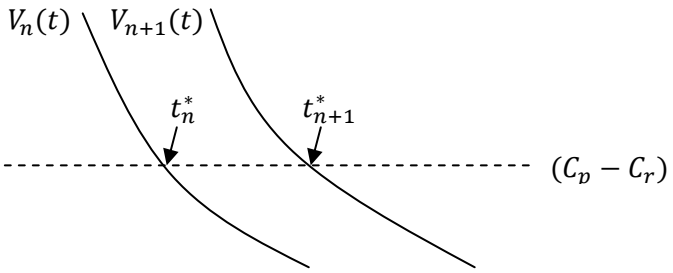

Figure 1: Optimal crash policy.

The structure of the optimal crash policy relies on the monotonicity property of $V_n(t)$. Therefore, the following lemma that $V_n(t)\ \uparrow n\ \downarrow t$ needs to be proved.

Lemma 3.2 $V_n(t)\ \uparrow n\ \downarrow t$.

Proof.

From the dynamic equation of $V_n(t)$ (18), for $V_{n+1}(t)$ and $V_n(t, t_{n+1}^*)$, where

$$V_n(t, t_{n+1}^*) = \int_t^{t_{n+1}^*} \bar{F}_n(s|t)\,[\gamma_n(s)((V_{n+1}(s) + C_r)\wedge C_p) + f'(t)]ds + \bar{F}_n(t_{n+1}^*|t)\varphi_n(t_{n+1}^*), \tag{30}$$

we have the following differential equations

$$V'_{n+1}(t) = -\gamma_{n+1}[(V_{n+2}(t) + C_r)\wedge C_p) - V_{n+1}(t)] - f'(t)$$

$$V'_n(t, t_{n+1}^*) = -\gamma_n[V_{n+1}(t) + C_r)\wedge C_p) - V_n(t, t_{n+1}^*)] - f'(t),$$

and we have

$$V'_{n+1}(t) - V'_n(t, t^*_{n+1}) \leq -\gamma_{n+1}[V_{n+1}(t) - V_n(t, t^*_{n+1})]. \quad (31)$$

Therefore, the boundary condition is now at t^*_{n+1}, where

$$V_{n+1}(t^*_{n+1}) = \varphi_{n+1}(t^*_{n+1}) > \varphi_n(t^*_{n+1}) \geq V_n(t^*_{n+1}). \quad (32)$$

Thus, $\forall\, t \in t^*_{n+1}$,

$$\left[\bar{F}_{n+1}(t)\left(V_{n+1}(t) - V_n(t, t^*_{n+1})\right)\right]' \leq 0, \quad (33)$$

$$(V_{n+1}(t) - V_n(t, t^*_{n+1})) \geq (\varphi_{n+1}(t^*_{n+1}) - \varphi_n(t^*_{n+1})) > 0, \text{ and} \quad (34)$$

$$V_{n+1}(t) \geq V_n(t, t^*_{n+1}) \geq V_n(t). \quad (35)$$

From the solution to the differential equations, we can see that $V'_n(t) < 0$. Thus, $V_n(t) \downarrow t \uparrow n$ can be observed directly from the differential equations.

The optimal crash policy is a control-limit policy with respect to the arrival time of each detected bug. There exists a series of increasing values of $\{t^*_n\}$ for any $n \geq 0$ such that the optimal crash is carried out at the first period when the n-th arrival time $S_n < t_n$. If $T_0 = t^*_n$, then the system is crashed at time 0.

As $V_n(t) \uparrow n \downarrow t$,

$$t^*_n = inf_t\{t | V_n(t) > (C_p - C_r I_{\{n>0\}})\} \quad (36)$$

is unique when exists.

3.4 Computational Algorithm

Based on the optimal policy and the boundary conditions, a computational algorithm has been developed. The algorithm is divided into four steps as below:

Step 1.

Let $x \sim Uniform\,[U_{MIN}, U_{MAX}]$. Starting from V^* with $\gamma^{U_{MAX}} = U_{MAX} g(t)$, the optimal policy is to release at age T^* such that

$$f'(T^*) = \gamma^{U_{MAX}} g(T^*)(C_R - C_r). \tag{37}$$

Therefore, there exists T^*, S^*, such that

for $t < S^*$, $V^*(t) = C_p$, crash immediately;

for $S^* < t < T^*$, solution to differential equation

$$\left(V^*(t)\right)' = -\gamma^{U_{MAX}}(t)\left[(V^*(t) + C_r) \wedge C_p - V^*(t)\right] - f'(t); \tag{38}$$

for $T^* < t$, $V^*(t) = \varphi^*(t) = C_r U_{MAX} \bar{G}(t)$.

$V^*(t)$ is the optimal cost function of a degenerated system with $X \equiv U_{MAX}$. For such a system, the optimal policy is either to crash at starting time *t* for $t \leq S^*$, or to release the system at T^* for $t > S^*$.

Step 2.

For a large N, let

$$V_N^N(t) = V^*(t)$$
$$W_N^N(t) = V^*(t) - \varphi^{U_{MAX}}(t) \tag{39}$$

Step 3.

For $n = (N-1):-1:1$,

(1) The optimal release time

$$T_n^* = \inf\{t | f'(t) = \gamma_n(t)\left((C_R - C_r) + |W_{n+1}(t)| \vee \left[\varphi_{n+1}(t) - \left(C_p - C_r\right)\right]\right)\} \tag{40}$$

(2) V_n^N is computed by differential equation

$$V'_{n+1}(t) = -\gamma_{n+1}[(V_{n+2}(t) + C_r) \wedge C_p) - V_{n+1}(t)] - f'(t) \tag{41}$$

(3) The optimal crash time

$$t_n^* = \inf\{t | V_n^N(t_n^*) \geq C_p - C_r I_{\{n>0\}}\} \tag{42}$$

(4) $W_n^N(t) = V_n^N(t) - \varphi_n(t)$

Step 4.

Stop after $n = 0$.

4. NUMERICAL EXAMPLE

The computational algorithm developed in the previous section is based on optimal policy and the boundary conditions. In this section, the algorithm is illustratively demonstrated by a numerical example, implemented in Matlab (See Appendix for Matlab Code). The numerical example assumes the following parameters:

- Crashing Penalty Cost, $C_p = 2000$.
- Cost of fixing a bug after release, $C_R = 200$.
- Cost of fixing a bug before release, $C_r = 15$.
- Cost of testing during period $[0, t]$, $f(t) = 20t$.
- Number of iteration, $N = 200$.
- Probability density function $g(t)$ of bug occurrence time follows an exponential distribution with parameter $\mu = 0.1$.
- The Random parameter X is considered to be uniformly distributed between 0 (U_{MIN}) and 200 (U_{MAX}).

The above parameters have been used in the demonstration of both sub-optimal and optimal policies. The first two plots (Figures 2 and 3) illustrate the expected total cost $V_n(t)$ over time, which is the value function under a fixed release time. The set of curves start from V_n, and follows a similar pattern until V_o is reached.

The difference between the penalty cost C_p and the cost of fixing each bug before release C_r is $(2000 - 15 =) 1985$. The intersection of the $(C_p - C_r)$ line and each of the $V_n(t)$ plots furnishes the crash threshold, shown in Figure 4 (blue curve). The crash threshold, plotted as the emergence of the number of bugs over time (days), represents the optimal crash times, t_n^*. The region on the left side of the crash threshold curve is simply the crash zone. This means, the

software is crashed if the number of bugs is too high, and the sample test path enters the crash threshold.

The lower bound of $V_n(t)$ defines the release policy, which is to release the software at time (days) 40.94. The release threshold, also shown in Figure 4 (red line), represents the optimal release times, T_n^*. The area on the right side of the release threshold is the release zone. The region between the crash and release thresholds represents the testing zone. A sample path is shown in Figure 4 (green line) (also shown in Figure 9) to demonstrate the occurrences of bugs during testing. It should be noted that only one bug occurs at a time.

Figures 5 and 6 illustrate the expected total cost $V_n(t)$ over time (days), under the optimal policies. The cost function in this case indicates a wider range of release times. Again, the intersection of the $(C_p - C_r)$ line and each of the $V_n(t)$ plots provides the crash threshold (represents the optimal crash times, t_n^*), shown in Figure 7. Similar to Figure 4, the region on the left side of the threshold in Figure 7 is the crash zone.

The release policy is defined at the lower bound of $V_n(t)$. The release threshold, shown in Figure 8, represents the optimal release times, T_n^*. The crash and release thresholds (from Figures 7 and 8) are compared in Figure 9. Here, the area between the crash and release thresholds also represents the testing zone.

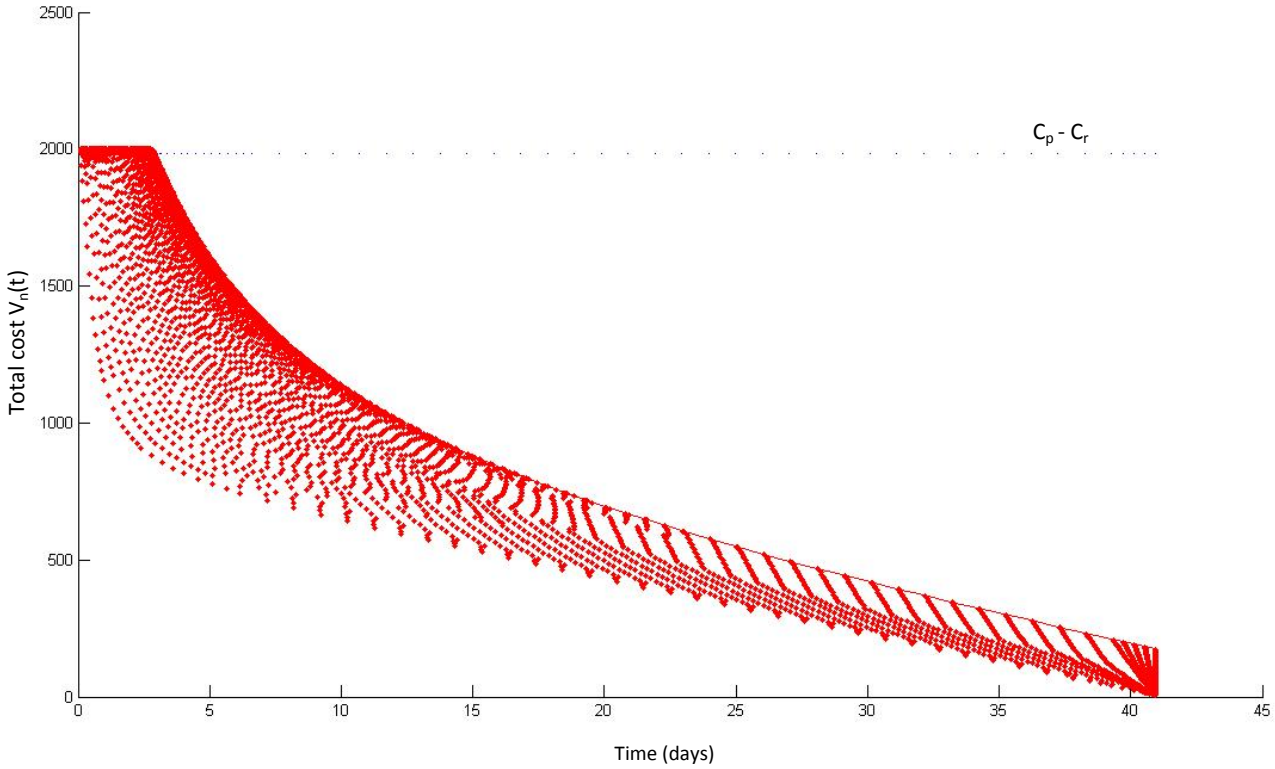

Figure 2: Total cost $V_n(t)$ over a fixed release time (under sub-optimal policies).

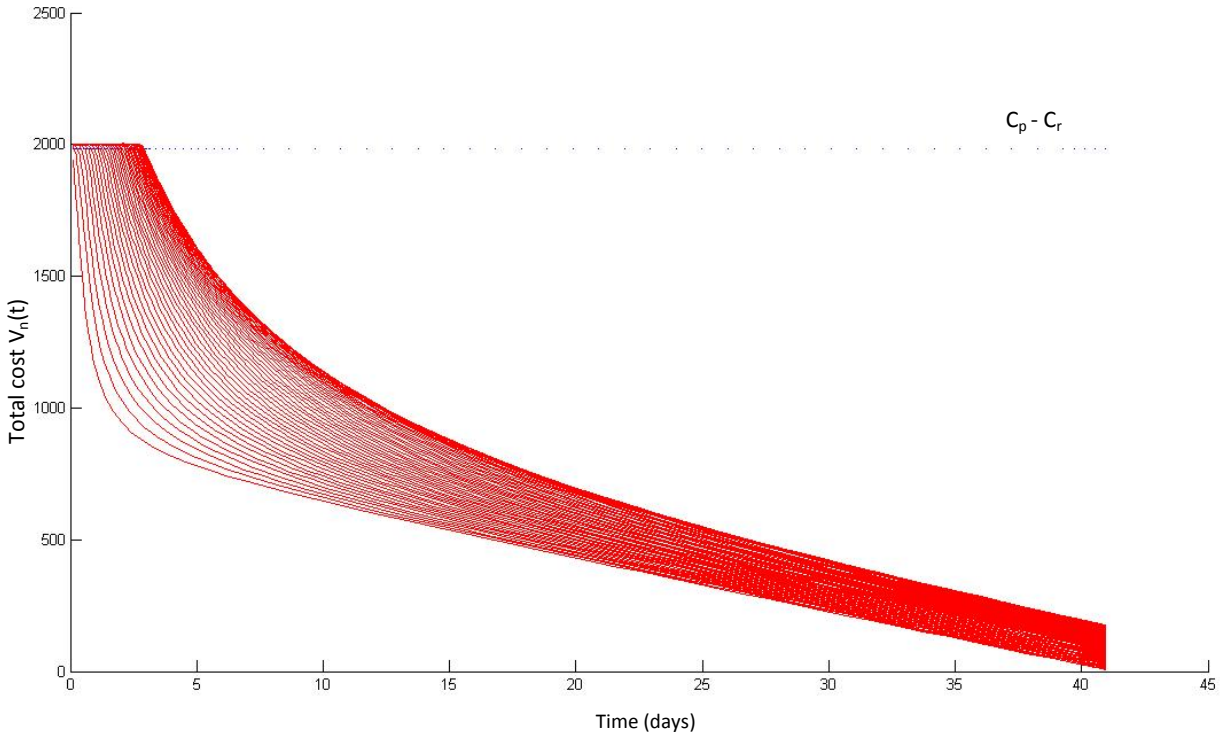

Figure 3 (dots connected): Total cost $V_n(t)$ over a fixed release time (under sub-optimal policies).

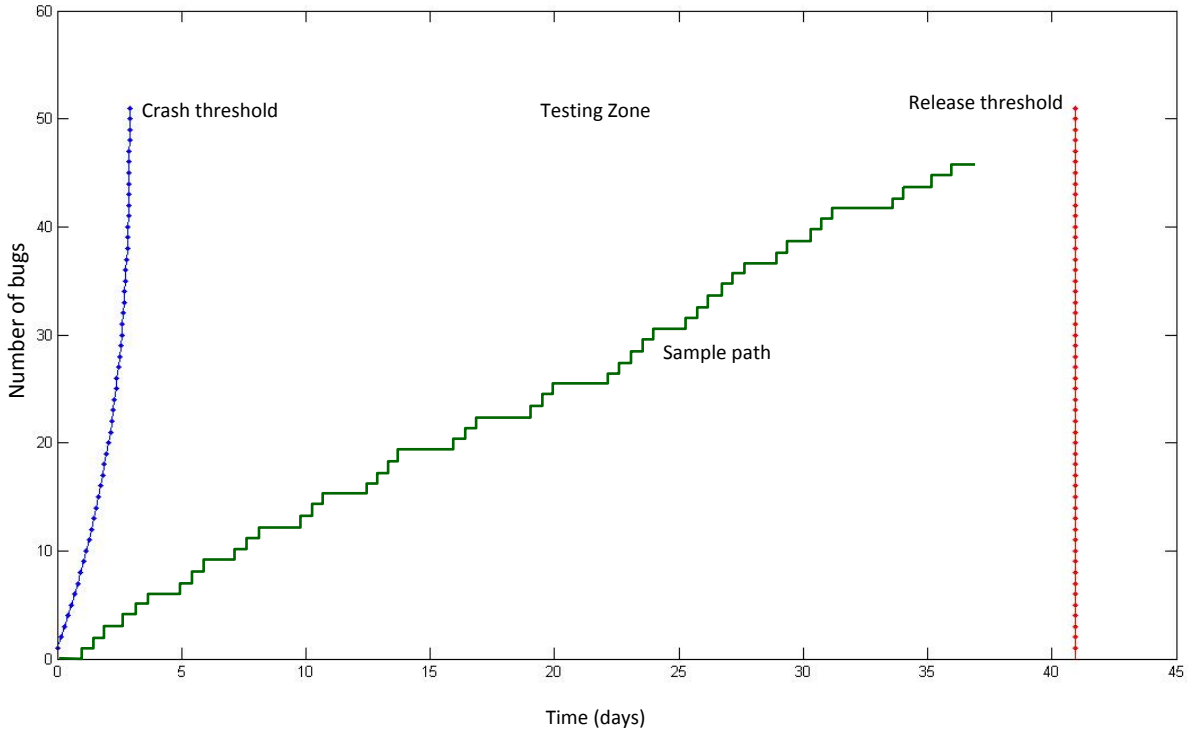

Figure 4: Crash and release thresholds under sub-optimal policies.

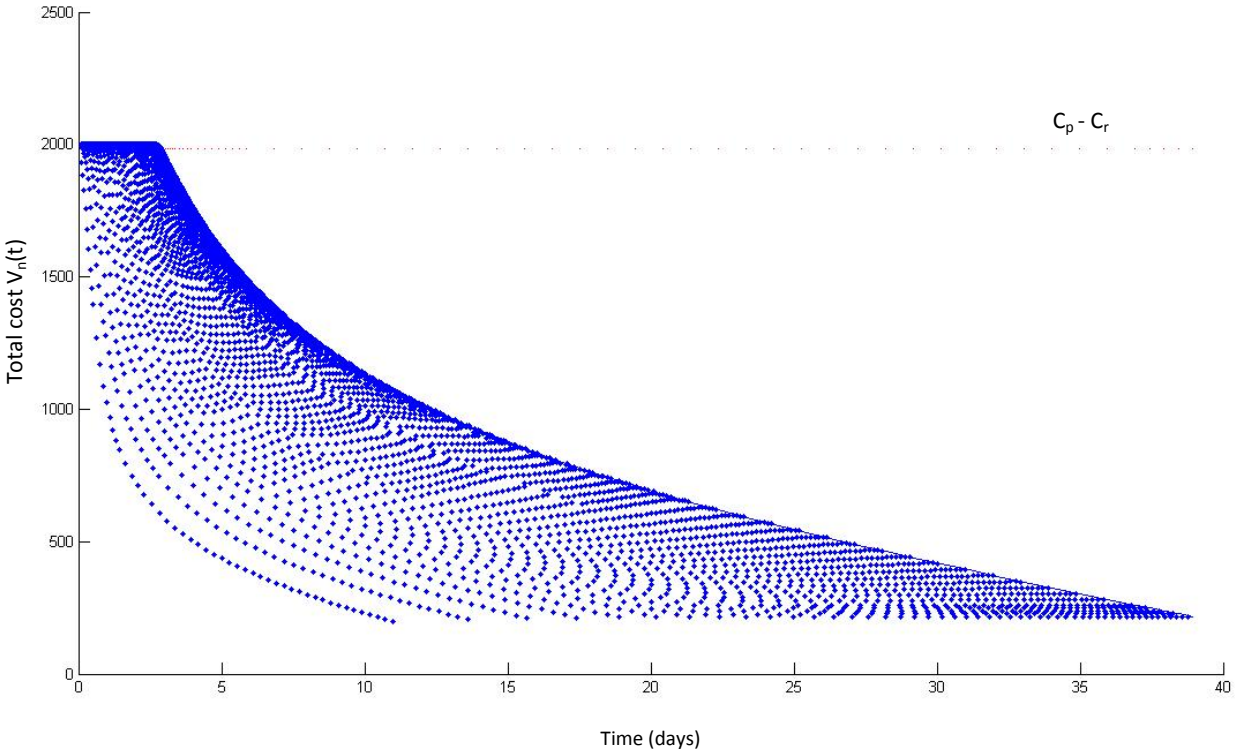

Figure 5: Total cost $V_n(t)$ under the optimal policies.

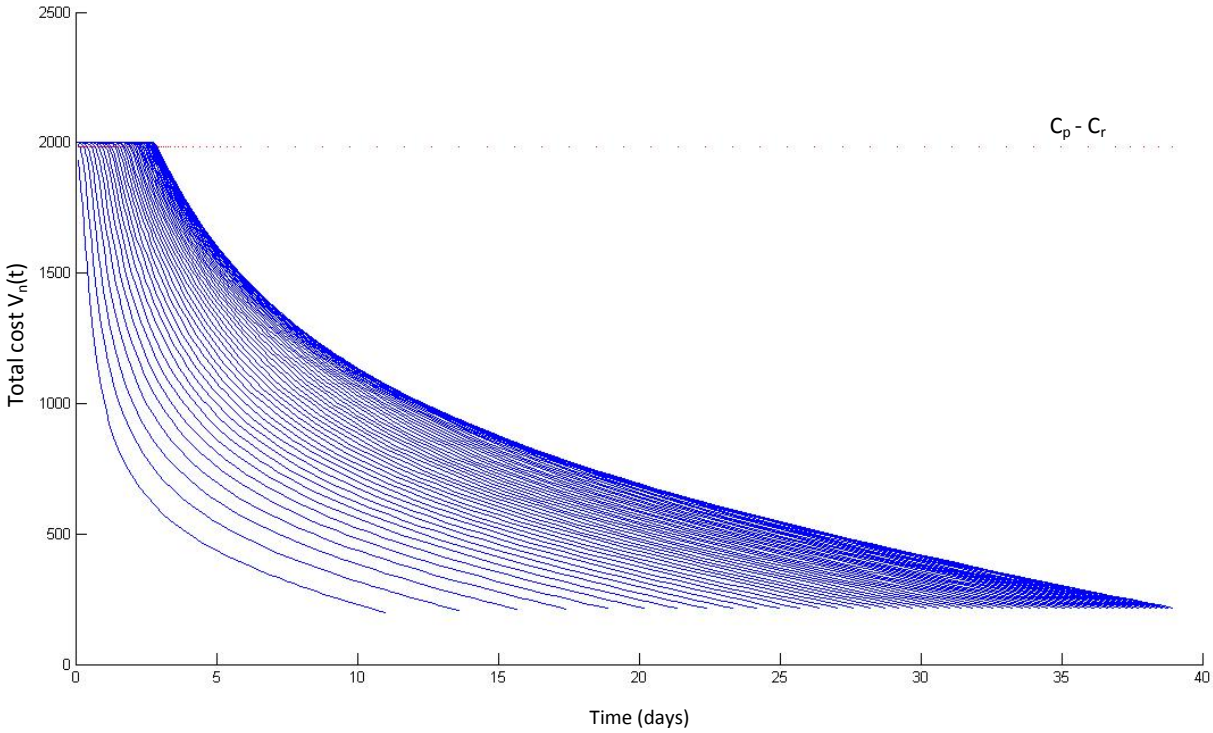

Figure 6 (dots connected): Total cost $V_n(t)$ under the optimal policies.

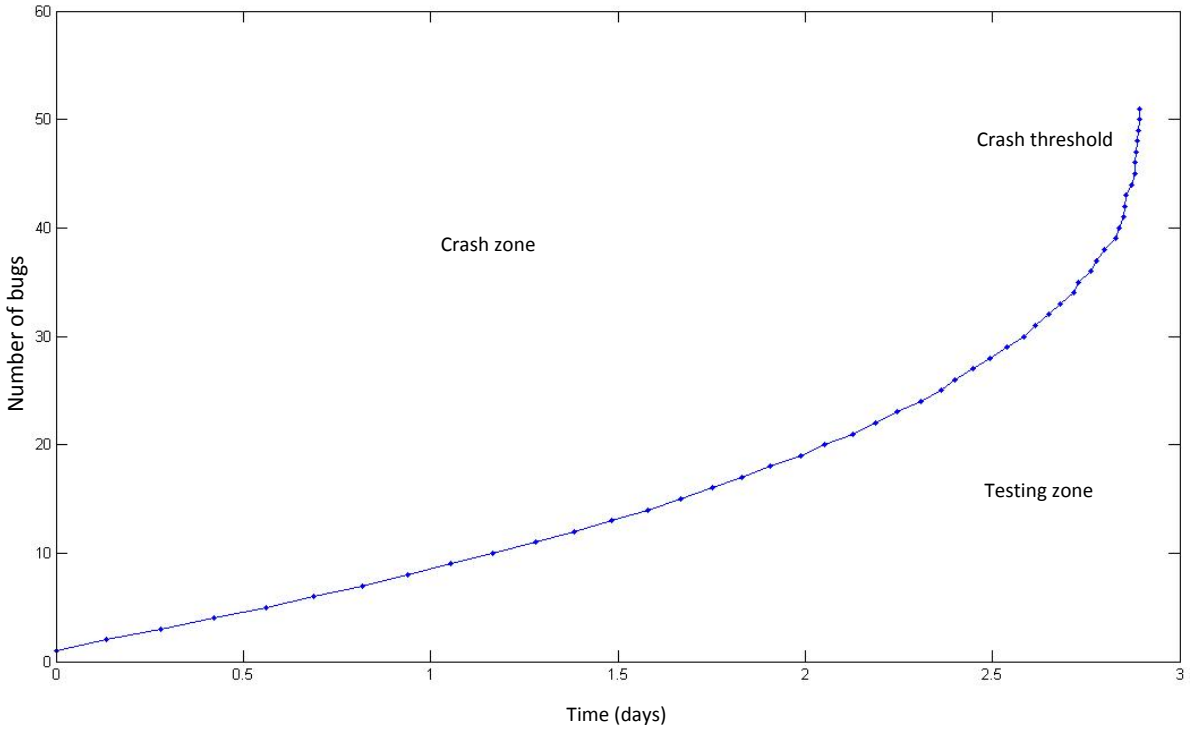

Figure 7: Crash threshold using the optimal policies.

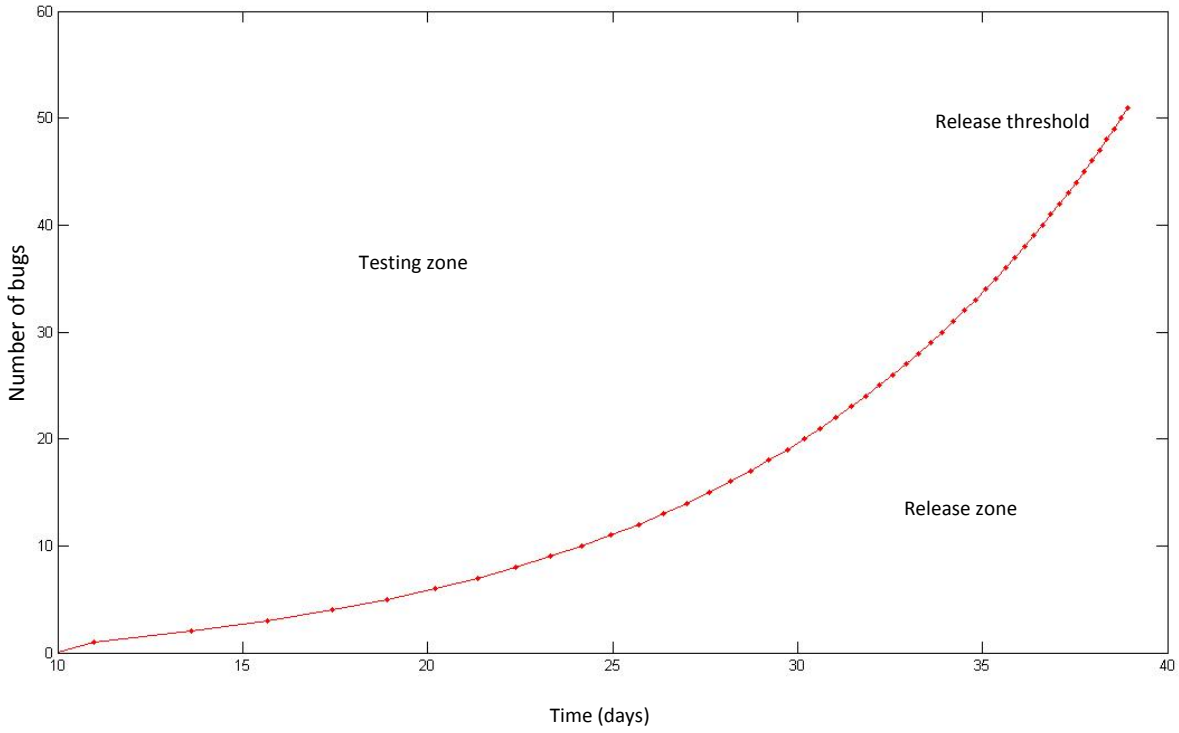

Figure 8: Release threshold using the optimal policies.

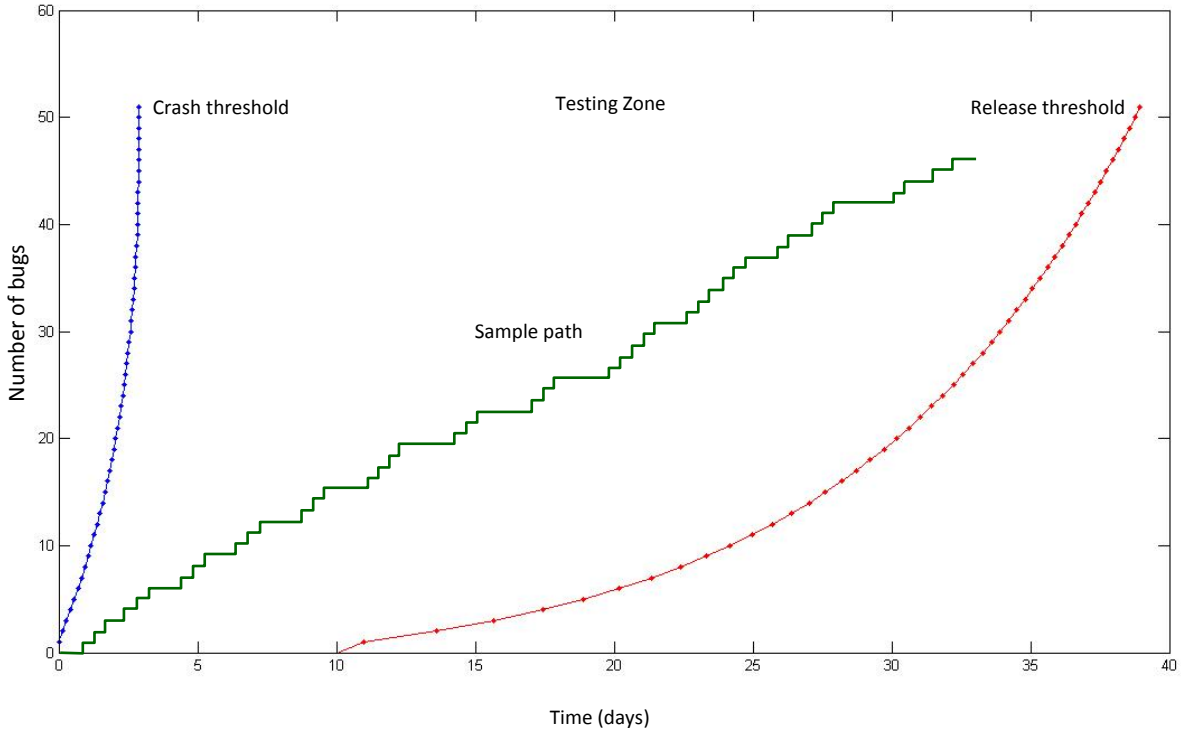

Figure 9: Optimal crash and release thresholds.

5. CONCLUSION

In this research, an enhanced software reliability model has been developed to demonstrate the optimal stopping time for software testing. The new optimal stopping formulation suggests the most favorable time to discontinue testing. The model exhibits both crash and release options that can be chosen at any time during the software testing phase, in order to avoid wasting effort in the development process.

The optimal crash policy contains a simple control limit structure with monotonicity properties. This enables early termination of testing, if the reliability of the software is low. The optimal release policy also contains a control limit structure with monotonicity properties, which allows the release option to be considered immediately if the software is found to be reliable enough. The dynamic policies have been shown to reduce testing time, which consequently minimizes the cost of testing.

The policies established in this research can be applied to any individual or integrated software modules. This means, smaller modules can be independently tested for reliability before they are incorporated into a complete package. The integrated package of many small modules can again be tested using the optimal policies to ensure completeness and better quality of the developed software.

The optimal policies will allow software developers to crash unreliable software before major costs have been incurred, or to release sufficiently reliable software prior to initially projected deadline. In a competitive software industry nowadays, implementing the optimal policies will give software developing companies a competitive advantage, by allowing them to cut down on developmental cost and to release the product before their competitors in the market.

Many industrial and commercial processes are governed by innovative software these

days, and it is becoming increasingly important for software companies to develop reliable software. Future research could be directed toward the development of further general debugging models, where self-generating processes are of great interest, and optimal stopping formulation is likely to be applicable.

REFERENCES

1. Ballmer, S., (2006), “The new IT investments powering productivity and growth”, Gartner Symposium/ ITxpo; http://www.gartner.com.

2. Biafore, B., (2006), "About project crashing (shortening a project schedule)", Microsoft Office Online, Microsoft Press; http://office.microsoft.com/en-us/help/HA100364 161033.aspx.

3. Chow, Y-S., Robbins, H., and Siegmund, D., (1991), “Great expectations: The theory of optimal stopping”, Mineola, NY: Dover.

4. Dalal, S. R., and Mallows, C. L., (1988), “When should one stop testing software”, *Journal of the American Statistical Association*, **83**(403), 872-879.

5. Dalal, S. R., and Mallows, C. L., (1990), “Some graphical aids for deciding when to stop testing software”, *IEEE Journal of Selected Areas in Communications*, **8**(2), 169–175.

6. Elliot, R. J., (1982), “Stochastic calculus and applications”, Springer, New York.

7. Forman, E. H., and Singpurwalla, N. D., (1977), “An empirical stopping rule for debugging and testing computer software”, *Journal of the American Statistical Association*, **72**(360), 750-757.

8. Forman, E. H., and Singpurwalla, N. D., (1979), “Optimal time intervals for testing on computer software errors”, *IEEE Transactions on Reliability*, **28**, 250–253.

9. Goel, A. L., and Okumoto, K., (1979), “Time-dependent error detection rate model for software reliability and other performance measures”, *IEEE Transactions on Reliability*, **28**(3), 206–211.

10. Huang, C., Lyu, M. R., and Kuo, S., (2003), “A unified scheme of some nonhomogenous Poisson process models for software reliability estimation”, *IEEE Transactions on Software Engineering*, **29**(3), 261-269.

11. Humphrey, W. S., (1989), “Managing the software process”, Addison-Wesley, Reading, MA.

12. Inoue, S., and Yamada, S., (2007), “Generalized discrete software reliability modeling with effect of program size”, *IEEE Transactions on Systems, Man, and Cybernetics – Part A: Systems and Humans*, **37**(2), 170-179.

13. Jelinski, Z., and Moranda, P., (1972), “Software reliability research”, Statistical Computer Performance Evaluation, W. Freiberger (ed.), Academic Press, 465-484.

14. Jiang, X., Knapp, G., and Barkman, M., (2005), “A dynamic software testing model with release and crash options”, Louisiana State University.

15. Kapur, P. K., and Garg, R. B., (1989), "Cost–reliability optimum release policies for a software system under penalty cost", *International Journal of Systems Science*, **20**, 2547–2562.

16. Koch, H. S., and Kubat, P., (1983), "Optimal release time of computer software", *IEEE Transactions on Software Engineering*, **9**(3), 323–327.

17. Kuo, L., and Yang, T. Y., (1996), "Bayesian computation for nonhomogeneous Poisson process in software reliability", *Journal of the American Statistical Association*, **91**(434), 206-211.

18. Lyu, M. R., (1996), "Handbook of software reliability engineering", McGraw-Hill.

19. Makis, V., and Jiang, X., (2003), "Optimal replacement under partial observations", *Mathematics of Operations Research*, **28**(2), 382-394.

20. Musa, J. D. and Okumoto, K., (1984), "A logarithmic Poisson execution time model for software reliability measurement", *Proceeding 7th International Conference on Software Engineering*, Orlando, Florida, 230-238.

21. Musa, J. D., Iannino, A., and Okumoto, K., (1987), "Software reliability – Measurement, prediction, application", New York: McGraw-Hill.

22. Ohba, M., (1984), "Software reliability analysis models", *IBM Journal of Research and Development*, **28**(4), 428–443.

23. Okumoto, K., and Goel, A. L., (1980), "Optimum release time for software systems based on reliability and cost criteria", *Journal of System Software*, **1**(4), 315–318.

24. Pan, J., (1999), "Software reliability", Dependable Embedded Systems, Carnegie Mellon University; http://www.ece.cmu.edu/~koopman/des_s99/sw_reliability.

25. Ross, S. M., (1985), "Software reliability: The stopping problem", *IEEE Transactions on Software Engineering*, **11**(12), 1472–1476.

26. Schick, G. J., and Wolverton, R. W., (1978), "An analysis of competing software reliability models," *IEEE Transactions on Software Engineering*, **SE**-4, 104-120.

27. Shanthikumar, J. G., and Tufekci, S., (1983), "Application of a software reliability model to decide software release time", *Microelectronics and Reliability*, **23**(1), 41-59.

28. Thayer, T. A., Lipow, M., and Nelson, E. C., (1976), "Software reliability study", Rep. RADC-TR-76-238.

29. Tohma, Y., Tokunaga, K., Nagase, S., and Murata, Y., (1989), "Structural approach to the estimation of the number of residual software faults based on the hyper-geometric distribution", *IEEE Transactions on Software Engineering*, **15**(3), 345–355.

30. Wikipedia, the free encyclopedia, (2007), “Software reliability”, Reliability Engineering; http://en.wikipedia.org/wiki/Reliability_engineering.

31. Xie, M., (1991), “Software reliability modeling”, Singapore: World Scientific Publisher.

32. Yamada, S., Ohba, M. and Osaki, S., (1984), “S–shaped software reliability growth models and their applications”, *IEEE Transactions on Reliability*, **33**(4), 289–292.

33. Yamada, S., Narihisa, H., and Osaki, S., (1984), “Optimum release policies for a software system with a scheduled delivery time”, *International Journal of Systems Science*, **15**, 905–914.

34. Yamada, S., and Osaki, S., (1985), “Cost-reliability optimal release policies for software systems”, *IEEE Transactions on Reliability*, **34**(5), 422-424.

35. Zheng, S., (2002), “Dynamic release policies for software systems with a reliability constraint”, *IIE Transactions*, **34**(3), 253-262.

APPENDIX: MATLAB CODE

Module 1: VntSolution_Alternative_Full.m - Computes $V_n(t)$ under a fixed release time.

```
global N Cp CR Cr Ct mu UMax UMin t V N_Bugs TN

parameters_Uniform;

N=50;

TMax=40.9434;

N_Bugs=N+1;

gtd=mu*exp(-mu*TMax);
Gtd=1-gtd/mu;

for i = 1:N+1
   Vntd(i)= CR*FuncEXnt(i,TMax)*(1-Gtd);
end

hold on;

j=N_Bugs;
if j==N+1
   [t_n,V_n] = ode45('dVntdt_Alternative', [TMax 0.1], Vntd(j));
   V=V_n;
   t=t_n;
   plot(t, V,'r-')
   X=Cp-Cr;
   plot(t, X, 'b-')
end

for j=N:-1:1
   N_Bugs=j
   [t_n,V_n] = ode45('dVntdt_Alternative', [TMax 0.1], Vntd(j));
   for i=1:size(V)
      if V(i)<Cp
      else
         V(i)=Cp;
      end
   end

   V_temp1=zeros(size(t_n));
   V_temp2=zeros(size(t_n));

   for i=1:size(t_n)
```

```
        V_temp1(i)=V_n(i);
        V_temp2(i)=spline(t,V,t_n(i));
    end

    V=zeros(size(t_n));
    V=min(V_temp1, V_temp2);
    t=t_n;
    plot(t, V,'r-')
end
```

Module 2: VntSolution_Alternative_Full2.m - Computes $V_n(t)$ under the optimal policies.

```
global N Cp CR Cr Ct mu UMax UMin t V N_Bugs TN

beginTime=clock;

parameters_Uniform;

N=50;

Crash_Times=zeros(1,N+1);

TMax=40.9434;

N_Bugs=N+1;

TN=zeros(1,N+1);

FindTN;

TN(N_Bugs)=min(TN(N_Bugs), TMax);

gtd=mu*exp(-mu*TN(N_Bugs));
Gtd=1-gtd/mu;

j=N_Bugs

    Vntd(j)= CR*FuncEXnt(j,TN(j))*(1-Gtd);

hold on;

if j==N+1
    [t_n,V_n] = ode45('dVntdt_Alternative', [TN(j) 0.1], Vntd(j));
    V=V_n;
    t=t_n;
    X=Cp-Cr;
    plot(t, X, 'r-')
```

```
    plot(t, V,'b-')
end

for j=N:-1:0
    N_Bugs=j
    FindTN;
    TN(N_Bugs)=min(TN(N_Bugs), TMax);
    gtd=mu*exp(-mu*TN(N_Bugs));
    Gtd=1-gtd/mu;
    Vntd(j)= CR*FuncEXnt(j,TN(N_Bugs))*(1-Gtd);
    [t_n,V_n] = ode45('dVntdt_Alternative', [TN(N_Bugs) 0.1], Vntd(j));
    for i=1:size(V)
        if V(i)<Cp
        else
            V(i)=Cp;
        end
        if V(i)<Cp-Cr
              Crash_Times(j)=t(i)*(V(i+1)-(Cp-Cr))/(V(i+1)-V(i))+t(i+1)*(1-(V(i+1)-(Cp-
Cr))/(V(i+1)-V(i)));
        end
    end
    V_temp1=zeros(size(t_n));
    V_temp2=zeros(size(t_n));
    for i=1:size(t_n)
        V_temp1(i)=V_n(i);
        V_temp2(i)=spline(t,V,t_n(i));
    end
    V=zeros(size(t_n));
    V=min(V_temp1, V_temp2);
    t=t_n;
L=min(V_n);
plot(t, V,'b-')
end

RunTime=clock-beginTime
```

Module 3: dVntdt_Alternative.m - Computes the derivative.

```
function y  = dVntdt_Alternative(w,z)
global N Cp Cr Ct CR mu UMax t V N_Bugs

if N_Bugs==(N+1)
    y=-FuncEXnt(N_Bugs,w)*mu*exp(-mu*w)*(min(z+Cr,Cp)-z)-Ct;
else
    V_NPlus1=spline(t,V,w);
    y=-FuncEXnt(N_Bugs,w)*mu*exp(-mu*w)*(min(V_NPlus1+Cr,Cp)-z)-Ct;
end
```

Module 4: FuncEXnt.m - Computes $E(X|N(t) = n)$.

```
function y=FuncEXnt(n, t)
global N Cp CR Cr Ct TD mu UMax UMin

Gt=1-exp(-mu*t);

y=(n+1)/Gt*(gammainc(UMax*Gt, n+2)-gammainc(UMin*Gt,
n+2))/(gammainc(UMax*Gt, n+1)-gammainc(UMin*Gt, n+1));
```

Module 5: FindTN.m - Computes the sub-optimal and optimal release/crash times.

```
global t V UMax Ct Cr CR mu N_Bugs TN TMax

xU=TMax;
xL=0;
x=xU;

Det=xU-xL;

for k=1:30

if N_Bugs==N+1
   y1=min(CR*FuncEXnt(N_Bugs+1,x)*exp(-mu*x), (Cp-Cr));
   y2=CR*FuncEXnt(N_Bugs+1,x)*exp(-mu*x); %

else
   y1=min(spline(t, V, x), (Cp-Cr));
   y2=CR*FuncEXnt(N_Bugs+1,x)*exp(-mu*x);
end

y=y2-y1+(CR-Cr);
Det=Det/2;
check=y*FuncEXnt(N_Bugs,x)*mu*exp(-mu*x)-Ct;

if check <0
   xU=xU-Det;
   x=xU;
else
   xU=xU+Det;
   x=xU;
end

end

if N_Bugs>0
   TN(N_Bugs)=x;
```

```
    y=y2-y1+(CR-Cr);
end
```

Module 6: parameters_Uniform.m – Defines the uniform parameters.

```
global Cp CR Cr Ct TD mu UMax UMin N

Cp=2000;
CR=200;
Cr=15;
Ct=20;
mu=0.1;
UMax=200;
UMin=0;
N=200;
```

VITA

Tanvir Khan is a native of Dhaka, Bangladesh. He has received a Bachelor of Science in Electrical Engineering degree with minor in mathematics, from Louisiana State University – Baton Rouge in 2004. During his undergraduate career, he has worked in the LSU Middleton Library, and the Department of Computer Science, and has interned with IBM Bangladesh. Prior to his completion of undergraduate studies, he has obtained the Louisiana Engineer Intern License from Louisiana Professional Engineering and Land Surveying Board in 2004.

He is currently pursuing the Master of Science in Engineering Science degree, with concentrations in information technology and engineering, and information systems and decision sciences. During his graduate studies, he has worked as a Teaching and Graduate Assistant in the Departments of Construction Management and Industrial Engineering, and Accounting, and has interned with LSU Career Services as a Database Developer.